RADAR: Runtime Asymmetric Data-access Driven Scientific Data Replication

John Jenkins[1,2,3], Xiaocheng Zou[1], Houjun Tang[1], Dries Kimpe[2], Robert Ross[2], and Nagiza F. Samatova[1]

[1] North Carolina State University, Raleigh, NC 27695, USA,
{xzou2,htang4,nfsamato}@ncsu.edu,
[2] Argonne National Laboratory, Argonne, IL 60439, USA
{jenkins,kimpe,rross}@mcs.anl.gov,
[3] Corresponding author: jenkins@mcs.anl.gov

Abstract. Efficient I/O on large-scale spatiotemporal scientific data requires scrutiny of both the logical layout of the data (e.g., row-major vs. column-major) and the physical layout (e.g., distribution on parallel filesystems). For increasingly complex datasets, hand optimization is a difficult matter prone to error and not scalable to the increasing heterogeneity of analysis workloads. Given these factors, we present a partial data replication system called RADAR. We capture datatype- and collective-aware I/O access patterns (indicating logical access) via MPI-IO tracing and use a combination of coarse-grained and fine-grained performance modeling to evaluate and select optimized physical data distributions for the task at hand. Unlike conventional methods, we store all replica data and metadata, along with the original untouched data, under a single file container using the object abstraction in parallel filesystems. Our system can produce up to manyfold improvements in commonly used subvolume decomposition access patterns. Moreover, the modeling approach can determine whether such optimizations should be undertaken in the first place.

1 Introduction

In high-performance computing (HPC) systems and parallel filesystems such as PVFS [7], Lustre [33], and GPFS [32], the distribution of data across multiple storage devices is a difficult problem, which numerous works have been dedicated to solving. The combination of high dimensionality (multiple variables distributed in a spatiotemporal domain) and distributed requests over many processes complicates making an informed decision about how to place data in order to achieve high performance. The problem is exacerbated when noncontiguous access patterns are induced on storage, such as subvolume access. Even optimizations made to reduce or eliminate noncontiguous disk access, such as two-phase collective I/O [40], create new access patterns for which the data distribution may not be optimized.

Previous works have looked at data layout optimization in an HPC context in two general respects: modifying the logical layout of data with the goal of producing specialized data organizations for a specific usage (e.g., range-query processing on scientific data [23, 12, 19]) and optimizing the physical distribution of datasets to better match the mapping of process requests to I/O servers [37, 52, 38], either in place or as separate entities in storage, and with varying degrees of adaptability. However, these works have some combination of the following potential problems: modified logical formats introduce both interoperability concerns and difficulties related to manual management of the custom format; works that provide multiple data layouts or replicate data in multiple formats rely on creating directories/files for each, leading to a large number of files to process any time the dataset is used; and the distribution formats are either fixed or optimize for a single metric (e.g., disk thrashing via DiskSim [3], requests to a single segment of file).

To mitigate problems, we present a model-driven, adaptive layout optimization framework, called RADAR, using direct parallel filesystem semantics. Our layout optimization is based on partial replication, allowing a controllable increase in dataset sizes in exchange for I/O performance optimization. Furthermore, as opposed to previous works, which fix either the regions of data to replicate or the replication format, we allow variability in both. In particular, we present the following contributions:

1) Adaptive replica management policy. Given a set of I/O access patterns, our replica layout manager (Section 2.1) uses an I/O performance modeling approach to (1) create replicas with varied layouts for performance optimization of input access patterns and (2) to rank replicas for inclusion under storage-limited scenarios. Furthermore, our approach can gracefully handle imbalances in both server loads and client loads, using performance modeling to account for the former and distribution heuristics to account for the latter. Using a prototype MPI-IO driver, we show that our method is effective at accelerating common subvolume decomposition tasks, showing multifold speedups under many scenarios.

2) Single-container, nonintrusive dataset storage. All replica data and metadata are stored alongside the original, unchanged data in a single file container, achieved through direct object-storage semantics (Section 2.2). Distribution of replica data among a fixed set of replica objects is enabled through a combination of sparse-file capabilities and a object-slice-based allocation scheme.

3) Datatype- and collective-aware MPI-IO tracing. As an enabling technology, we develop a tracer capable of collecting full logical I/O requests with low overhead at the MPI-IO-level (Section 2.3). It is configurable to collect either precollective or postcollective optimizations (or both).

Our paper is organized as follows. Section 2 describes the framework, including necessary background. Section 3 presents our experiments. Section 4 examines related work. Section 5 briefly summarizes our conclusions.

2 Method

The system workflow, in which access patterns are garnered from applications and replicated data layouts are created to optimize for those access patterns, is realized by a number of components, as shown in Figure 1. First we develop a datatype-aware, collective-aware I/O trace layer that captures I/O requests. Then we process traces using a pattern analyzer, outputting access patterns of interest, such as strided access. Our replica layout manager ingests these patterns and, along with previously generated pat-

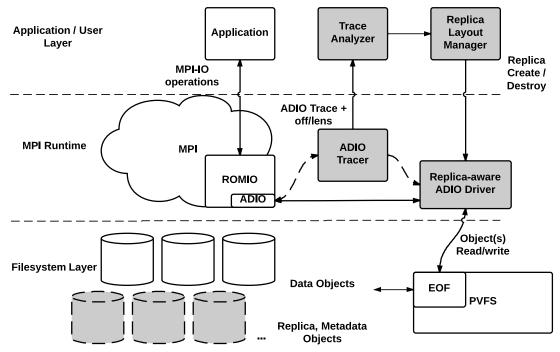

Fig. 1. RADAR components, across the I/O software stack. The shaded figures delineate our contributions.

terns, determines what data to replicate and in what format. Our replica-aware, object-storage-based ADIO implementation matches I/O requests to replications, redirecting the subsequent EOF object operations.

Since the bulk of our methodology lies in the layout manager and works independent of the method of replica distribution and access pattern generation, we first discuss it in Section 2.1. Next, we describe the data management policies employed by our method in Section 2.2, followed by the tracing and trace-analysis components in Section 2.3. The replica-aware ADIO driver is discussed in Section 2.4.

2.1 RADAR Layout Manager

The goal of RADAR's layout manager component is to create replicas with a layout that improves I/O performance under a given set of access patterns. To provide a framework capable of doing so, we use two strategies. First, to optimize in the presence of concurrent accesses, we generate replicas for *time-delimited pattern sets*. Second, to quickly generate and evaluate candidate replica sets, we adopt a *two-phase* performance-modeling approach, using a coarse-grained performance model to quickly produce and select candidate replica sets and using a fine-grained performance model to compute the estimated performance difference against the original data layout. The following sections discuss the individual components.

Pattern Preprocessing Preprocessing of the patterns is driven by our optimization goals: create a replica enabling efficient access of the pattern, while being aware of concurrent system operations. The latter has been examined in previous work by converting all accesses into log-structured, effectively creating a one-to-one process-to-server mapping [1, 52]. Here we are looking at flexible distribution among multiple servers for read optimization, rather than write optimization.

Each pattern in our analyzed traces has starting and ending times. Given these and a value δ, the patterns are partitioned into *buckets*, each corresponding to a time window of length δ. Each bucket is then considered a single entity for the purposes of performance modeling and optimization. Since each bucket need not be sorted in our method, the overall process is linear in the number of patterns.

Replica Generation and Ranking – Performance Modeling We use a simple, constant-time performance model to generate candidate replicas and a more involved model to give a relative performance comparison between the original data layout and candidate layouts. This design decision is made in order to quickly generate replicas for testing, while retaining the ability to evaluate against unbalanced access patterns with respect to either the amount of data requested per process or the amount of data processed by each I/O server.

Preliminaries Our models use latency/bandwidth measurements over both network and storage, assuming serialization of requests at the node level (both client and server) and requests to storage. Table 1 shows the relevant variables. Furthermore, we make a few simplifying assumptions across both models that, while harmful to general-purpose high-accuracy performance prediction, still allow us to make valid measures for comparative purposes over time-gated accesses in a manner that is computationally reasonable. First, we assume no pipelining of network and storage operations, insulating us from false positives arising from slightly different access schedules but giving a pessimistic view of system capabilities. Second, resource contention is measured through the aggregation of request latencies and, in the fine-grained performance model, through penalization terms on nodes based on the number of distinct requests. This approach misses some phenomena observed in real runs or in full system/subsystem simulators [3], such as disk head thrashing.

Table 1. Performance Model System Parameters

System parameters	
n	I/O servers
ℓ_{net}	I/O request (network) latency
b_{net}	Network per-byte transfer time
ℓ_{sto}	I/O request (disk) latency
b_{sto}	Storage per-byte transfer time
r_s	Local storage readahead
Per-pattern-set inputs	
$\mathbb{P}$	Set of access patterns with process mappings
p	I/O participants (clients)
m	I/O participants per node
Coarse-grained model parameters	
B	Total request size across all patterns (derived)
n_p	Servers contacted per client (input)
r	Average request size per client per server $= B/(pn_p)$

Coarse-Grained Model The coarse-grained model is a generalization of the cost model created by Song et al. [37] to optimize accesses under the following characteristics: uniform access sizes, perfect access distribution among servers corresponding to PVFS data layouts, and single time of issuance across all processes. This model, while not created for general-purpose I/O modeling, has proven useful for HPC applications with regular access patterns and is appropriate for driving our replica placement method, given that we are in full control of replica placement and can produce such regular accesses. First we discuss the model and generalization; then we discuss how we use the model to find effective replica layouts.

The cost model by Song et al. has four separate costs that are summed to find the final result: T_e, the establishment time for all network operations; T_x, the time to transfer all request data across the network; T_s, the "startup" time for all storage accesses; and T_{rw}, the read/write time for all storage accesses. The original model computes these costs based on a number of specific, fixed layouts mapping process requests to servers (see [37] for more details). We observe that the parameter being varied across each of the models is the *servers contacted per client*. Making this an explicit variable n_p allows us to collapse the equations into a single set:

$$T_e = \max(mn_p, \lceil \frac{pn_p}{n} \rceil)\ell_{net} \tag{1}$$

$$T_x = \max(mn_p, \lceil \frac{pn_p}{n} \rceil)rb_{net} \tag{2}$$

$$T_s = \lceil \frac{pn_p}{n} \rceil \ell_{sto} \tag{3}$$

$$T_{rw} = \lceil \frac{pn_p}{n} \rceil rb_{sto}. \tag{4}$$

Coarse-Grained Model Usage Given the definition of the coarse-grained model, we derive a simple replica creation process, using the following strategy. Assume the underlying accesses are regular and uniform, compute $\min_{n_p}(T_e + T_x + T_s + T_{rw})$, and then resolve any load balances by over/underprovisioning replica striping across the servers. After computing B and an average m, simply calculate model values for $n_p \in \{1, 2, \ldots, n\}$, and choose the minimum. Next, perform the logical striping under the assumption that r is the actual request size per client. Then, compute for each pattern which servers its data resides on. An example mapping is shown in Figure 2. The intuition behind this layout heuristic is that patterns with balanced accesses will be optimized as normal and that overprovisioning for unbalanced accesses with larger relative sizes will be made up for by underprovisioning for accesses with smaller sizes, mapping degree of concurrency to relative access size.

Fine-Grained Model The fine-grained model shares similarities with the coarse-grained model, using latency/bandwidth modeling at both the network and the storage levels to generate an overall cost. However, whereas the coarse-grained model makes some significant assumptions about access characteristics, we need a more robust model capable of capturing load imbalance. Our approach consists of the following two steps, with the underlying steps of mapping each pattern in the pattern set to its respective client

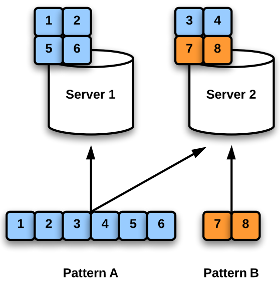

Fig. 2. Access pattern over and under provisioning based on model optimization on balanced accesses (for $n_p = 2$).

process/node and set of contacted servers: (1) compute a localized T'_e, T'_s and T'_{rw} for each server, and (2) calculate the total time to receipt T'_t for each client node based on the server calculations, with the maximum among them being the total request time.

The computation of T'_s and T'_{rw} is relatively straightforward, with T'_s being the number of noncontiguous blocks (measured at a page granularity and taking into account readahead) accessed times the storage access latency and T'_{rw} being the total size of all requests to the server times the inverse of the bandwidth. For T'_{rw}, we additionally adjust the performance to account for readahead: if two consecutive requests are within a readahead window (default 128 KB on Linux), then the disk latency cost is avoided at the cost of consuming the bytes separating the two requests.

The computation of T'_e and T'_t are more nuanced, since we must consider both access latency and wait times for request/response receipt. Because computing these wait times exactly would require a known access schedule (as well as a more architecturally accurate simulation), we instead compute an approximate. For T'_e, we compute the network latency times the number of incoming requests, with a *penalization term* ϵ_e. For this, we take the node contacting the server with the maximum number of outgoing requests, and we assume that the request to the server occurs after all of its other requests. Similarly for T'_t, the penalization term ϵ_t is computed by assuming that the data requested by the given node is issued after the server's access schedule, for the contacted server with the largest load.

2.2 EOF Data Management

Background – PVFS and EOF Recently, the "end-of-files" (EOF) [13] extension to the Parallel Virtual File System (PVFS) [7] was created to expose the object storage abstraction directly into the client space, presenting a file as a set of distinct, physically distributed objects that applications forward requests directly to, as opposed to a more implicit mapping via a striping function. For example, dataset metadata can be forwarded to a single object, while the data itself can be assigned distinct objects based on timesteps, variables, and so forth. We use EOF to achieve our layout optimization goals.

RADAR Object Layout The EOF object layout is such that nearly all RADAR data components exist under a single filename. The original dataset is stored in unchanged form, striped by some distribution across multiple objects. We include a file metadata object since distribution in EOF is relegated to the user. For RADAR, at file create time we allocate a number of replica objects equal to the number of data objects. We do this both for practicality reasons (limits on per-file concurrency) and for semantic reasons (currently, EOF cannot dynamically add or remove objects from a file container). A replica metadata object is used to store the mapping of replicas to object locations. We place the set of generated access patterns processed by RADAR in a dedicated object for the results to persist across multiple application runs. Note that the trace and trace analysis output currently exist outside the MPI/EOF file container, although these can be integrated with ease.

Replica Object Storage Strategy One problem with using a shared set of replica objects is how to distribute multiple replicas with heterogeneous distributions in the manner the layout manager instructs. To this end, we exploit sparse-file capabilities in the local filesystems employed by PVFS. Essentially, sparse files do not store blocks not written to asides from metadata—a block can represent data at, for example, a gigabyte offset without additionally having blocks representing all previous offsets.

We divide all objects into allocation units we call *object domains* (ODs), an example of which is shown in Figure 3. An OD corresponds to the full set of replica objects, spanning a per-object address space with fixed, large-granularity sizes. Each replication is placed within a single OD. To avoid biasing replica placement towards one object or another, replicas are assigned starting objects in round-robin order. In the figure, for example, the replica shown begins addressing at the leftmost object, while the next replica created will begin addressing at the next leftmost object.

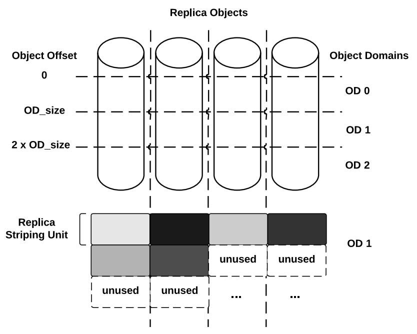

Fig. 3. Allocation units in RADAR (“object domains,” or ODs), and replica layout in an OD.

2.3 I/O Tracer and Analyzer

Our tracing methodology is achieved through MPI-IO, implemented by using the underlying ADIO [41] interface in ROMIO [42], the MPI-IO implementation in MPICH. This strategy allows access efficiently to both MPI datatypes used in I/O calls and the result of underlying collective calls. The output of the tracer includes all ADIO calls, as well as all per-process offset/length pairs, in a plain-text format. Compression methodologies such as inline pattern analysis or off-the-shelf compressors will be explored in future work. More details about our tracing methodology can be found in our previous work [20].

To gather the desired access patterns, we built a variant of the IOSig trace analysis software [4, 50]. We similarly use a template matching approach, but, as our tracer works at the ADIO level and additionally processes datatypes, the processing of the traces has been rewritten. For more discussion pertaining to access pattern categorization and discovery, see [4, 50]. For this paper, the specific access patterns we gather are contiguous access patterns (sequential access of a large space with fixed or average-size request sizes) and k-d strided access patterns (accesses that differ in offset by a fixed value, or stride), both of which are common in HPC I/O workloads, typically corresponding to accesses along spatio-temporal domains or across multiple variables.

2.4 Replica-Aware ADIO Driver

The responsibilities of the RADAR ADIO driver are to interface with EOF, maintain the semantically varying sets of objects, and remap I/O requests into the replica space, as appropriate. Asides from the remapping portion, the rest of the processing is relatively simple, corresponding to loading/distributing file and replica metadata and driving some RADAR-specific operations, such as carrying out the replication.

Given a set of replicated data layouts and a set of I/O requests (e.g., logical file offset/length pairs via a call to `MPI_File_read`), reading from replicas occurs in two steps: finding applicable replicas to read and choosing among the resulting candidates.

When finding replicas that can satisfy a given request, a linear-time matching approach over all replicas is undesirable. Furthermore, using spatial data structures such as interval or R-trees requires flattening strided patterns into their individual contiguous blocks, leading to extremely large search structure sizes. Hence, we use a binning approach, similar to generating an *inverted index* [48, 53] over the file, mapping logical regions of file (the bins) to a list of replicas overlapping with the regions to prune the search space. Additionally, to help prevent worst-case linear behavior (all replicas overlap with a bin), we employ a one-element history, under the assumption that consecutive I/O requests are highly likely to map to the same distinct replicas multiple times. Note that for read requests, we consider only replicas that contain the full file offset/length requested, since splitting up the requests further to disparate physical locations would likely reduce performance.

Once we have a set of candidate replicas to choose from, the next task is to select which replica among the set to read from. Note that this choice is nonexistent for writes, since all replicas would need to be updated. Furthermore, complicating the decision is that the information available is inherently local—individual processes cannot have a full system view. Therefore, we use a simple heuristic we call *smallest containing block* (SCB). The idea behind SCB is that of *specialization*: we consider replicas with a finer granularity to be more specialized than those without, and we believe they should be prioritized in the replica selection process. Generally, then, replicas over strided data will more than likely be selected over replicas over contiguous data, since each of the strided data structures will be more sparse.

3 Experimental Evaluation

All experiments were run on the Fusion cluster at Argonne National Laboratory. Each node contains two quad-core Intel Xeon processors at 2.53 GHz with 32 GB RAM, and nodes are connected by InfiniBand QDR. Each node in Fusion contains local hard-disk storage (250 GB IBM iDataPlex). Additionally, our implementation of RADAR is based on MPICH 3.0.2 and PVFS2 2.8.1, patched with EOF. Because of issues with InfiniBand support for PVFS on Fusion, both MPI communication and PVFS client-server communication are performed via TCP over InfiniBand.

Since we use a modified version of PVFS and since each node in Fusion has local storage, we assign a subset of the nodes to serve as PVFS I/O servers and use the remaining as I/O clients. We use eight I/O servers in all experiments. Hence, each server initially contains 8 GB of data, striped using 1 MB blocks.

Table 2 shows the performance model parameters we gathered via microbenchmarks on Fusion. We use the BMI `pingpong` utility in PVFS to gather network performance through PVFS, where BMI (Buffered Message Interface) is PVFS's client/server communication interface. We use simple read benchmarking via collocating a PVFS client with a server to gather storage performance parameters. Note that the microbenchmark result for disk bandwidth is much lower than expected: the bandwidth when not going through PVFS is 90 MB/s. We were unable to eliminate this discrepancy, but we believe it to be a result of internal threading and buffering overheads on the PVFS server.

Table 2. Performance Model Variables

System parameters		
n	8	I/O servers
r_s	128 KB	Local storage readahead
ℓ_{net}	32.9μs	I/O request (network) latency
ℓ_{sto}	6.20ms	I/O request (disk) latency
b_{net}	0.00112μs (867 MB/s)	Network per-byte transfer time
b_{sto}	0.0212μs (44.98 MB/s)	Storage per-byte transfer time

For our inverted list acceleration structure for replica lookup, we divided the file into 1,024 bins, each covering a 64 MB extent of data.

3.1 Benchmarks

We evaluate our layout optimization work within the context of four different multi-dimensional array decompositions, shown in Figure 4: row-wise (distribute volume by contiguous plane), column-wise (distribute volume by non-contiguous plane), block-wise (distribute volume by 3D subvolume), and timestep-wise (distribute single sub-volume from a range of timesteps). The row-wise decomposition induces contiguous patterns at each process, while the remaining decompositions induce multidimensional strided patterns at each process. For all experiments, we used a subvolume of (time, X, Y, Z) dimensions $128 \times 256 \times 256 \times 256$ in row-major order, each element of which is a 32-byte structure (e.g., four C `doubles`). The total size of this dataset is 64 GB. Note that these access patterns are a superset of the access patterns exhibited by several well-known benchmarks such as MPI-Tile-I/O [27], IOR [18], and PIO-bench [34], all of which perform accesses with regular (single- or multidimensional) strides.

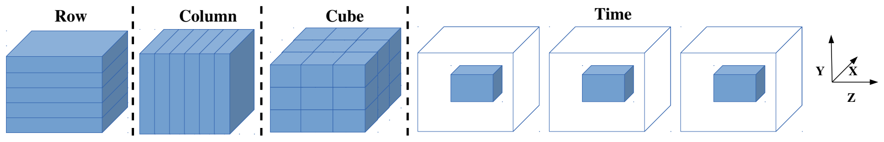

Fig. 4. Subvolume decompositions used in our evaluation (contiguous in order Z, Y, X, time).

3.2 Decomposition Performance

We test each decomposition using independent I/O with all processes on each node participating. As shown in our previous work [20], a single participant-per-node configuration exhibits similar behavior on our test system, and replication on collective I/O access patterns (e.g., large, contiguous patterns) shows no improvement, although modeling is capable of capturing this case.

Figure 5 shows performance under the different decompositions both before and after RADAR data replication. For these runs, we synchronize prior to running the decomposition and calculate bandwidth with respect to the maximum elapsed time for each individual read. We note a few points about these experiments:

1. All decompositions except the time-based decomposition decompose the same overall data size of 2 GB (four timesteps of 512 MB volumes). Thus, with an increasing number of clients, the average request size decreases, and the number of requests increases, leading to potentially less efficient access when not using collective I/O.
2. The time-based decomposition defines a fixed-size subvolume for each client to read of size 64 MB. As the number of clients increase, the per-client requests remain the same, leading to an increase in the total request size.
3. The cube decomposition divides the volume into perfect cubes ($1, 8, 27, 81, 125,$ etc.) no less than the number of clients, and clients are assigned multiple blocks to read, resulting in varying request granularities based on the number of clients. For

instance, a four-client run will divide the subvolume into eight blocks and assign two blocks to each client. This approach can lead to both load imbalance (processes can be oversubscribed blocks compared with others) and varied access patterns because of the possibility of multiple smaller blocks combining into a single, large, contiguous block.

The top left of Figure 5 shows the time-based decomposition. Without replication, the aggregate performance is far below peak performance because of the noncontiguous accesses. The use of data reorganization through replication enables high performance across the spectrum, although tapering off once I/O servers begin processing requests from multiple clients.

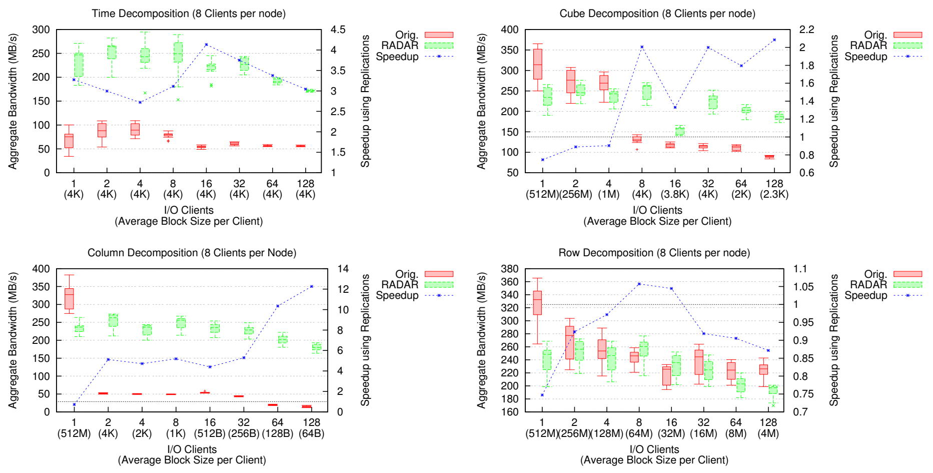

Fig. 5. Decomposition results with and without replication via RADAR.

Figure 5 shows the cube-based decomposition. This decomposition results in block sizes of high variance with a changing number of clients, which is a significant factor in the overall performance. In the figure, the performance implications can easily be seen between the four-client and eight-client decomposition for the nonreplication case, and the eight-client and 16-client decomposition for the RADAR case. Regardless, the use of RADAR helps smooth out the performance characteristics as a result of making the strided accesses contiguous per client and over/underprovisioning of replicas based on load. Additionally, for the small-client case, we notice performance regressions between the no-replication and replication case. We are currently unable to diagnose this difference; the generated layout by RADAR is the same, and the I/O driver follows largely the same code path.

The bottom left of Figure 5 shows the column-based decomposition performance. This represents a pathological I/O pattern, as seen by the average contiguous block sizes. Hence, performance without collective optimizations or RADAR is far worse than any of the other decompositions as the number of clients increase. RADAR can greatly improve performance over the original data layout, although it also tapers off somewhat for increasing client counts.

The bottom right of Figure 5 shows the row-based decomposition performance, representing the "best case" for parallel I/O without reorganization: large, contiguous, non-overlapping blocks. Here, since the storage is the primary bottleneck and block sizes are very large, RADAR is not shown to have any benefit.

3.3 Model Verifications

We now look at how the performance modeling approach compares with the performance shown in Section 3.2. The goal of the performance models is to show whether a specific data layout can be improved by a modified data layout via replication. Note that this goal is different from strict performance accuracy: here, the primary measurement of interest is the accuracy of relative performance between two layouts (one with replicas, one without). Since the models do not perform full system simulation, they are unsuitable for general-purpose performance prediction.

Figure 6 shows the results, comparing the model-derived performance of both the original layout and the layout under replication with the median of the performance shown in Section 3.2. We additionally show the estimated performance using the coarse-grained model corresponding to the best layout. In general, the "best layout" found corresponds to the heuristic of spreading each pattern's data across as many servers as possible until overlap occurs, in which case the distribution contracts accordingly.

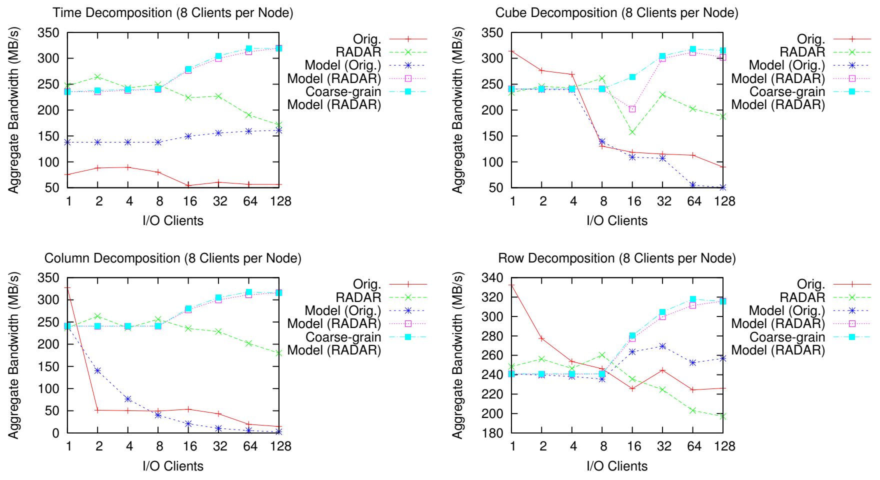

Fig. 6. Model results against median empirical performance (8 clients per node).

Overall, the model results capture some degree of performance difference between the original layout and the candidate replica set layout. Large, contiguous accesses are shown to have little difference between both the original and the replicated layouts, implying that creating the replica set would result in minimal, if any, gain. Smaller, noncontiguous accesses, on the other hand, are correctly shown to have a large degree of benefit by the models.

A few nontrivial aspects of the test system and software prevent the models from performing more accurately. First, the model assumes that additional nodes and servers correspond to additional network resources to draw upon. As seen in the time-based decomposition (fixed access sizes per client), this is not the case. Hence, more accurate, architecture-specific network modeling is needed. Furthermore, the cost models generally overestimate the performance: we believe this overestimation to be the cause of our readahead simulation being optimistic in its ability to effectively cache pages without interfering with normal system performance, as well as overheads such as buffering that were not accounted for.

4 Related Work

4.1 Replication in Storage-I/O Systems

Data replication in storage systems is a well-researched topic in many domains. Many parallel/distributed filesystems, such as the Hadoop Distributed File System [35] and the Google File System [10, 26], built for task-centric, data-intensive workloads, as well as the Ceph filesystem [46], have data replication as a first-order feature. Local filesystem replication has also been explored in a performance context by reorganizing data to minimize rotational latency and maximize locality [2, 15].Additionally, hybrid methods are being explored in parallel filesystems without intrinsic replication support, such as a *shim* layer for PVFS allowing Hadoop-style workloads and replication [39].

Database systems also widely use replication, both for fault tolerance and as a performance optimizer. For instance, replicas can be created by using query history as a guide [28, 47] or in a more dynamic approach where replication/indexing occurs as queries are performed [17, 16].

The use of replication to ensure high availability and/or improve performance has also been explored through high-level libraries and I/O middleware. For MPI-based applications, works have shown that file block replication using the PMPI interface provides application-level I/O resiliency [36], while replicating data in different storage layouts can be used to improve performance for one-to-one, process-to-file configurations [37, 51]. Additionally, specific metrics can be optimized via replication and reorganization, such as minimizing disk head thrashing by examining local disk traces with DiskSim [3, 52].

4.2 Capturing and Detecting I/O Access Patterns

Many approaches have been developed to systematically derive system usage information from applications. For MPI-based applications, the MPI Parallel Environment (MPE) [8] provides full MPI event tracing, while mpiP [44] provides lightweight, statistical measures. The ScalaTrace family of MPI tracers focuses on compressed trace generation [29, 31, 45, 49], using histogram generation and a combination of intranode and internode trace compression. Dynamic instrumentation methods include automatically instrumenting at compile time through source code analysis [22], as well as runtime binary instrumentation through IOPin [21], based on the Pin [24] framework. For a "big-picture" view, Darshan [6, 5] focuses on *center-wide* usage patterns by combining local, subsystem metrics (such as the Sysstat [11] and fsstats [9] utilities) and application-level metrics (instrumented through POSIX and MPI-IO).

Once acceptable profiles or logs of application/system performance are gathered, they can be mined for emergent patterns. Statistical learning methods can be used in a general sense to capture high-level patterns such as block-to-block association [25, 30, 43, 2]. Recent methods have been developed specifically for HPC, again typically through the MPI/MPI-IO layers. Examples include strided pattern analysis for MPI prefetching [4] and pattern recognition for PLFS index compression under a checkpointing use-case [14]. The IOSig [4] trace analyzer converts I/O operations to compact and parameterized representations called I/O signatures using a *template matching* approach, which iteratively attempts to match specific patterns (e.g., regularly strided) to the sequence of I/O accesses.

5 Conclusion

Effective data distribution in large-scale analysis systems is an integral component of achieving high-performance I/O, especially in the presence of complex, noncontiguous workloads such as the volume decompositions we have presented. Through the tight coupling with a filesystem view of the data as a set of distinct objects, we were able to create arbitrary data layouts optimized for the access patterns induced on the dataset, all in a single container. RADAR is a promising step in the direction of automated specialization of data layouts based on application-specific needs and access patterns, providing both increased performance and an initial ability to reason about the "worth" of layouts for the purpose of marshaling usage of limited space for optimized data distributions.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

References

1. J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.
2. M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for self-optimizing storage systems. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 183–196, Berkeley, CA, USA, 2009. USENIX Association.

3. J. S. Bucy, J. Schindler, S. Schlosser, G. Ganger, and Contributors. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University Parallel Data Lab, 2008.
4. S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008*, pages 1–12. IEEE, 2008.
5. P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOC)*, 7(3):8:1–8:26, Oct. 2011.
6. P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *IEEE International Conference on Cluster Computing*, Cluster'10, pages 1–10, 2009.
7. P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.
8. A. Chan, W. Gropp, and E. Lusk. User's guide for MPE: Extensions for MPI programs. Technical Report ANL/MCS-TM-ANL-98/xx, Argonne National Laboratory, 2003.
9. S. Dayal. Characterizing HEC storage systems at rest. Technical Report CMU-PDL-09-109, Carnegie Mellon University Parallel Data Laboratory.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
11. S. Godard. Sysstat utilities home page. http://sebastien.godard.pagesperso-orange.fr/index.html.
12. Z. Gong, D. A. B. II, X. Zou, Q. Liu, N. Podhorszki, S. Klasky, X. Ma, and N. F. Samatova. PARLO: PArallel Run-time Layout Optimization for scientific data explorations with heterogeneous access patterns. In *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, Delft, The Netherlands, 2013.
13. D. Goodell, S. J. Kim, R. Latham, M. Kandemir, and R. Ross. An evolutionary path to object storage access. In *Proceedings of the Seventh Workshop on Parallel Data Storage*, PDSW '12, 2012.
14. J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. Discovering structure in unstructured I/O. In *Proceedings of the Seventh Workshop on Parallel Data Storage*, PDSW '12, 2012.
15. H. Huang, W. Hung, and K. G. Shin. Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 263–276, New York, NY, USA, 2005. ACM.
16. S. Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, University of Amsterdam, 2010.
17. S. Idreos, M. Kersten, and S. Manegold. Database cracking. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research*, CIDR'07, 2007.
18. Interleaved or random (IOR) parallel filesystem I/O benchmark.
19. J. Jenkins, I. Arkatkar, S. Laksminarasimhan, D. A. Boyuka II, E. R. Schendel, N. Shah, S. Ethier, C. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. F. Samatova. ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Transactions on Large Scale Data and Knowledge Centered Systems (TLDKS)*, 8220:95–114, 2013.
20. J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova. Parallel data layout optimization of scientific data through access-driven replication. Technical Report TODO, North Carolina State University, 2014.
21. S. J. Kim, S. W. Son, W.-k. Liao, M. Kandemir, R. Thakur, and A. Choudhary. IOPin: Runtime profiling of parallel I/O in HPC systems. In *7th Parallel Data Storage Workshop*, PDSW'12, 2012.

22. S. J. Kim, Y. Zhang, S. W. Son, R. Prabhakar, M. Kandemir, C. Patrick, W.-k. Liao, and A. Choudhary. Automated tracing of I/O stack. In *Proceedings of the 17th European MPI Users Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 72–81, Berlin, Heidelberg, 2010. Springer-Verlag.
23. S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 31:1–31:11, New York, NY, USA, 2011. ACM.
24. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.
25. T. M. Madhyastha and D. A. Reed. Learning to classify parallel input/output access patterns. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):802–813, Aug. 2002.
26. M. K. McKusick and S. Quinlan. GFS: Evolution on fast-forward. *Queue*, 7(7):10:10–10:20, Aug. 2009.
27. Parallel I/O benchmarking consortium. `http://www.mcs.anl.gov/research/projects/pio-benchmark/`.
28. S. Narayanan, U. Catalyurek, T. Kurc, V. S. Kumar, and J. Saltz. A runtime framework for partial replication and its application for on-demand data exploration. In *High Performance Computing Symposium, SCS Spring Simulation Multiconference*, HPC '05, 2005.
29. M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, Aug. 2009.
30. J. Oly and D. A. Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 147–155, New York, NY, USA, 2002. ACM.
31. P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 46–55, New York, NY, USA, 2008. ACM.
32. F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.
33. P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, 2003.
34. F. Shorter. Design and analysis of a performance evaluation standard for parallel file systems. Master's thesis, Clemson University, 2003.
35. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
36. S. W. Son, R. Latham, R. Ross, and R. Thakur. Reliable MPI-IO through layout-aware replication. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, SNAPI '11, 2011.
37. H. Song, Y. Yin, Y. Chen, and X.-H. Sun. A cost-intelligent application-specific data layout scheme for parallel file systems. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 37–48, New York, NY, USA, 2011. ACM.
38. H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang. A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In *Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM International Symposium on*, pages 414–423, 2011.
39. W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. On the duality of data-intensive file system design: Reconciling HDFS and PVFS. In *Proceedings of*

2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11, pages 67:1–67:12, New York, NY, USA, 2011. ACM.
40. R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, 1996.
41. R. Thakur, W. Gropp, and E. Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '96, pages 180–187, Washington, DC, USA, 1996. IEEE Computer Society.
42. R. Thakur, R. Ross, E. Lust, and W. Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.
43. N. Tran and D. A. Reed. Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):362–377, Apr. 2004.
44. J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 123–132, New York, NY, USA, 2001. ACM.
45. K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM.
46. S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
47. L. Weng, U. Catalyurek, T. Kurc, G. Agrawal, and J. Saltz. Servicing range queries on multidimensional datasets with partial replicas. In *IEEE International Symposium on Cluster Computing and the Grid*, volume 2 of *CCGrid '05*, pages 726–733. IEEE, 2005.
48. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.
49. X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Probabilistic communication and I/O tracing with deterministic replay at scale. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 196–205, Washington, DC, USA, 2011. IEEE Computer Society.
50. Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur. Boosting application-specific parallel I/O optimization using IOSIG. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 196–203, 2012.
51. Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In *IEEE International Symposium on Parallel and Distributed Computing*, IPDPS'13, pages 345–356, 2013.
52. X. Zhang and S. Jiang. InterferenceRemoval: Removing interference of disk access for mpi programs through data replication. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 223–232, New York, NY, USA, 2010. ACM.
53. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.